Part 2

Building an OAuth 2 environment

In this section, you'll get to build an entire OAuth 2.0 ecosystem from scratch, including the client, protected resource, and authorization server. We'll step through each of these components and see how all of the bits interact with each other during the process of implementing the authorization code grant type introduced in the previous section. With that thoroughly under our belts, we'll tackle several optimizations and variations on the OAuth 2.0 protocol including different client types and different grant types.

[redacted]

[redacted]

3 Building a simple OAuth client

This chapter covers

- Registering an OAuth client with an authorization server and configuring the client to talk to the authorization server
- Requesting authorization from a resource owner using the authorization code grant type
- Trading the authorization code for a token
- Using the access token as a bearer token with a protected resource
- Refreshing an access token

As we saw in the last chapter, the OAuth protocol is all about getting tokens to the client and letting the client use the access tokens to access protected resources on behalf of the resource owner. In this chapter, we'll build a simple OAuth client, use the authorization code grant type to get a bearer access token from an authorization server, and use that token with a protected resource.

> **NOTE** All of the exercises and examples in this book are built using Node.js and JavaScript. Each exercise consists of several components designed to run on a single system accessible from *localhost* on various ports. For more information about the framework and its structure, see appendix A.

[redacted]

3.1 Register an OAuth client with an authorization server

First things first: the OAuth client and the authorization server need to know a few things about each other before they can talk. The OAuth protocol itself doesn't care *how* this happens, only that it *does* happen in some fashion. An OAuth client is identified by a special string known as the client identifier, referred to in our exercises and in several parts of the OAuth protocol with the name `client_id`. The client identifier needs to be unique for each client at a given authorization server, and is therefore almost always assigned by the authorization server to the client. This assignment could happen through a developer portal, dynamic client registration (as discussed in chapter 12), or through some other process. For our example, we're using manual configuration.

Open up the `ch-3-ex-1` folder and run `npm install` from inside it. For this exercise, we'll be editing `client.js` and leaving both `authorizationServer.js` and `protectedResource.js` untouched.

Why a web client?

You may have noticed that our OAuth client is itself a web application running on a web server hosted by the Node.js application. The fact that our *client* is also a *server* can be confusing, but in the end it's fairly simple: an OAuth Client is always the piece of software that gets a token from the authorization server and uses that token with a protected resource, as we talked about in chapter 2.

We're building a web-based client here because it's not only OAuth's original use case but also one of the most common. Mobile, desktop, and in-browser applications can also use OAuth, but each of those requires slightly different considerations and processing to make them work. We'll cover each of those in chapter 6, paying special attention to what differentiates these cases from our web-based client here.

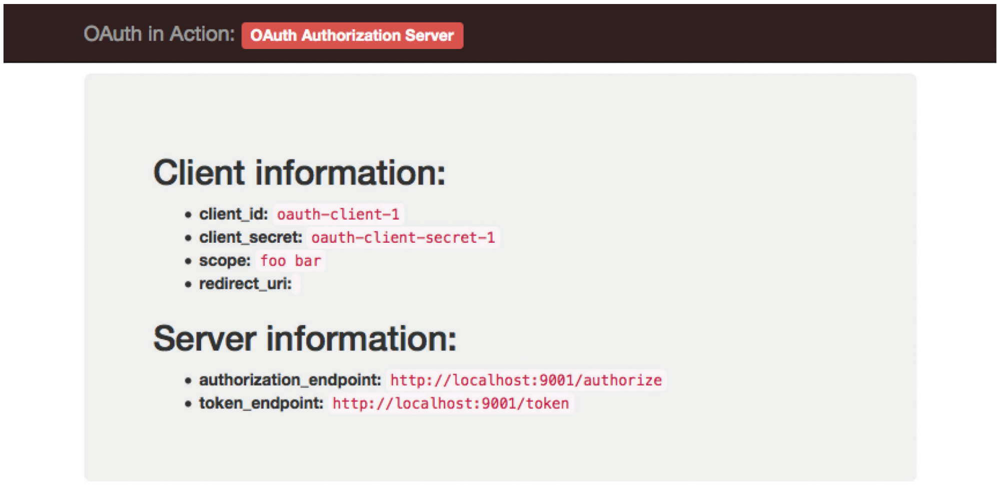
Our authorization server has assigned this client a `client_id` of `oauth-client-1`, (see figure 3.1) and now we need to transfer that information to the client software (to see this, look inside the top of the authorizationServer.js file and find the `client` variable at the top, or navigate to `http://localhost:9001/`).

Our client stores its registration information in a top-level object variable named `client`, and it stores its `client_id` in a field of that object named, unsurprisingly, `client_id`. We just need to edit the object to fill in the assigned `client_id` value:

```
"client_id": "oauth-client-1"
```

Our client is also what's known as a *confidential client* in the OAuth world, which means that it has a shared secret that it stores in order to authenticate itself when talking with the authorization server, known as the `client_secret`. The

[redacted]



Client information:

- client_id: oauth-client-1
- client_secret: oauth-client-secret-1
- scope: foo bar
- redirect_uri:

Server information:

- authorization_endpoint: http://localhost:9001/authorize
- token_endpoint: http://localhost:9001/token

Figure 3.1 The authorization server's homepage, showing client and server information

`client_secret` can be passed to the authorization server's token endpoint in several different ways, but in our example we will be using HTTP Basic. The `client_secret` is also nearly always assigned by the authorization server, and in this case our authorization server has assigned our client the `client_secret` of `oauth-client-secret-1`. This is a terrible secret, not only because it fails to meet minimum entropy requirements but also because now that we've published it in a book it's no longer secret. Regardless, it will work for our example, and we'll add it to our client's configuration object:

```
"client_secret": "oauth-client-secret-1"
```

Many OAuth client libraries also include a few other configuration options in this kind of object, such as a `redirect_uri`, a set of scopes to request, and other things that we'll cover in detail in later chapters. Unlike the `client_id` and `client_secret`, these are determined by the client software and not assigned by the authorization server. As such, they've already been included in the client's configuration object. Our object should look like this:

```
var client = {
   "client_id": "oauth-client-1",
   "client_secret": "oauth-client-secret-1",
   "redirect_uris": ["http://localhost:9000/callback"]
};
```

On the other side of things, our client needs to know which server it's talking to, and how to talk to it. In this exercise, our client needs to know the locations of both the authorization endpoint and the token endpoint, but it doesn't really need to know anything about the server beyond that. Our server's configuration information has been

[redacted]

stored on the top-level variable named `authServer`, and we've included the relevant configuration information already:

```
var authServer = {
  authorizationEndpoint: 'http://localhost:9001/authorize',
  tokenEndpoint: 'http://localhost:9001/token'
};
```

Our client now has all of the information that it needs to connect to the authorization server. Now, let's make it *do* something.

3.2 Get a token using the authorization code grant type

For an OAuth client to get a token from an authorization server, it needs to be delegated authority by a resource owner in one form or another. In this chapter, we'll be using an interactive form of delegation called the *authorization code grant type*, wherein the client sends the resource owner (which in our case is the end user at the client) over to the authorization server's authorization endpoint. The server then sends an authorization code back to the client through its `redirect_uri`. The client finally sends the code that it received to the authorization server's token endpoint to receive an OAuth access token, which it needs to parse and store. To take a look at all of the steps of this grant type in detail, including the HTTP messages used for each step, look back at chapter 2. In this chapter, we'll be concentrating on the implementation.

Why the authorization code grant type?

You may have noticed that we're focusing on one specific OAuth grant type: the authorization code. You may have already used other OAuth grant types (such as the implicit grant type or client credentials grant type) outside of this book, so why not start there? As we'll talk about in chapter 6, the authorization code grant type fully separates all of the different OAuth parties and is consequently the most foundational and complex of the core grant types that we'll cover in this book. All of the other OAuth grant types are optimizations of this one, suited for specific use cases and environments. We'll cover all of those in chapter 6 in detail, where you'll get a chance to adapt the exercise code to these other grant types.

We're going to stay in the `ch-3-ex-1` exercise that you've been building in the last section and expand its capabilities into a functional OAuth client. The client has been preconfigured with a landing page that starts off the authorization process. This landing page is hosted at the root of the project. Remember to run all three components simultaneously, each in its own terminal window as discussed in appendix A.

You can leave the authorization server and protected resource running throughout this exercise, but the client application will need to be restarted every time you make an edit in order to propagate your changes.

[redacted]

3.2.1 Sending the authorization request

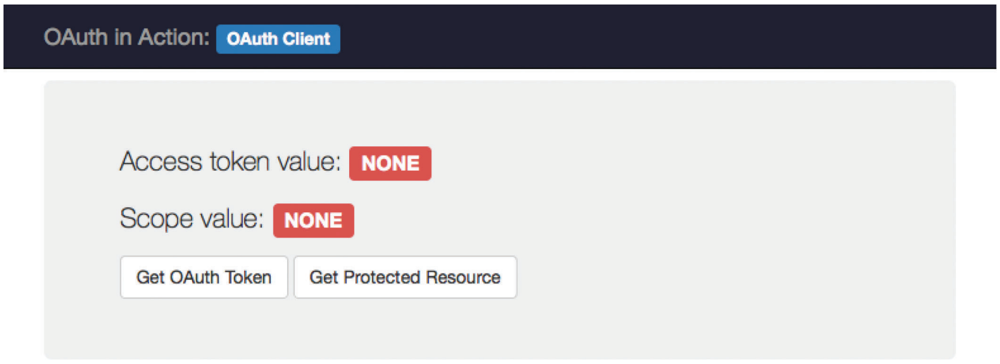
The homepage of the client application contains one button that sends the user to `http://localhost:9000/authorize`, and one button that fetches the protected resource (figure 3.2). We'll focus on the Get OAuth Token button for the moment. This page is served by the (currently empty) function:

```
app.get('/authorize', function(req, res){

});
```

To initiate the authorization process, we need to redirect the user to the server's authorization endpoint and include all of the appropriate query parameters on that URL. We'll build the URL to send the user to by using a utility function and the JavaScript `url` library, which takes care of formatting the query parameters and URL encoding values for you. We've provided the utility function for you already, but in any OAuth implementation you're going to need to properly build URLs and add query parameters in order to use front-channel communication.

```
var buildUrl = function(base, options, hash) {
   var newUrl = url.parse(base, true);
   delete newUrl.search;
   if (!newUrl.query) {
        newUrl.query = {};
   }
   __.each(options, function(value, key, list) {
        newUrl.query[key] = value;
   });
   if (hash) {
        newUrl.hash = hash;
   }

   return url.format(newUrl);
};
```

You can call this utility function by passing in a URL base and an object containing all the parameters you want added to the URL's query parameters. It's important to


OAuth in Action: OAuth Client

Access token value: NONE

Scope value: NONE

Figure 3.2 The client's initial state before getting a token

[redacted]

use a real URL library here, as throughout the OAuth process we might need to add parameters to URLs that already have them, or are formatted in strange but valid ways.

```
var authorizeUrl = buildUrl(authServer.authorizationEndpoint, {
   response_type: 'code',
   client_id: client.client_id,
   redirect_uri: client.redirect_uris[0]
});
```

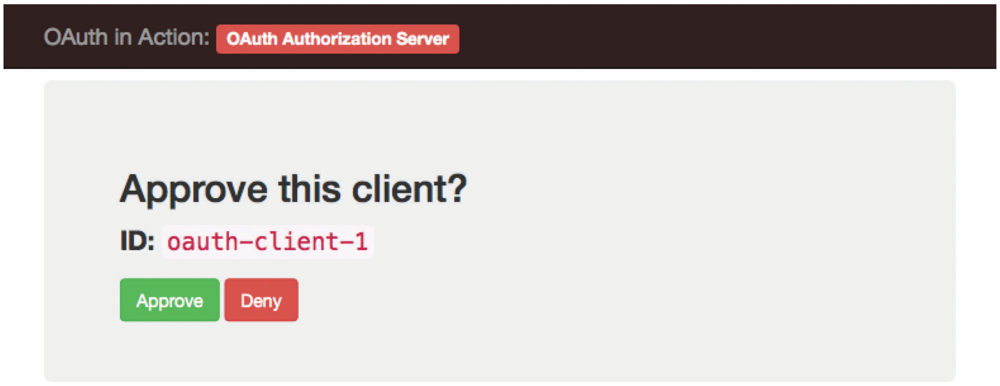
Now we can send an HTTP redirect to the user's browser, which will send them to the authorization endpoint:

```
res.redirect(authorizeUrl);
```

The `redirect` function is part of the express.js framework and sends an HTTP 302 Redirect message to the browser in response to the request on `http://localhost:9000/authorize`. Every time this page is called in our example client application, it will request a new OAuth token. A real OAuth client application should never use an externally accessible trigger mechanism such as this, but should instead rely on tracking internal application state to determine when a new access token is needed. For our simple exercise, we're OK with using an external trigger. When it's all put together, our final function looks like listing 1 in appendix B.

Now when you click the Get OAuth Token button on the client's main page, you should be automatically redirected to the authorization server's authorization endpoint, which should prompt you to authorize the client application (figure 3.3).

For this exercise, the authorization server is functionally complete, but we'll be digging into what's required to make it work in chapter 5. Click the Approve button, and the server will redirect you back to the client. Nothing particularly interesting will happen yet, so let's change that in the next section.



OAuth in Action: OAuth Authorization Server

Approve this client?

ID: oauth-client-1

Figure 3.3 The authorization server's approval page for our client's request

[redacted]

3.2.2 *Processing the authorization response*

At this point, you're back at the client application, and on the URL `http://localhost:9000/callback`, with a few extra query parameters. This URL is served by the (currently empty) function:

```
app.get('/callback', function(req, res){

});
```

For this part of the OAuth process, we need to look at the input parameters and read back the authorization code from the authorization server, in the `code` parameter. Remember, this request is coming in as a redirect from the authorization server, not as an HTTP response to our direct request.

```
var code = req.query.code;
```

Now we need to take this authorization code and send it directly to the token endpoint using an HTTP POST. We'll include the code as a form parameter in the request body.

```
var form_data = qs.stringify({
   grant_type: 'authorization_code',
   code: code,
   redirect_uri: client.redirect_uris[0]
});
```

As an aside, why do we include the `redirect_uri` in this call? We're not redirecting anything, after all. According to the OAuth specification, if the redirect URI is specified in the authorization request, that same URI must also be included in the token request. This practice prevents an attacker from using a compromised redirect URI with an otherwise well-meaning client by injecting an authorization code from one session into another. We'll look at the server-side implementation of this check in chapter 9.

We also need to send a few headers to tell the server that this is an HTTP form-encoded request, as well as authenticate our client using HTTP Basic. The Authorization header in HTTP Basic is a base64 encoded string made by concatenating the username and password together, separated by a single colon (:) character. OAuth 2.0 tells us to use the client ID as the username and the client secret as the password, but with each of these being URL encoded first.[1] We've given you a simple utility function to handle the details of the HTTP Basic encoding.

```
var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(client.client_id,
   client.client_secret)
};
```

[1] Many clients forgo URL encoding their client ID and secret, and some servers forgo doing the decoding on the far end. Since common client IDs and secrets consist of simple random sets of ASCII characters, this doesn't tend to cause problems, but for full compliance and support of extended character sets be sure to URL encode and decode appropriately.

[redacted]

We'll then need to wire that in with a `POST` request to the server's authorization endpoint:

```
var tokRes = request('POST', authServer.tokenEndpoint,
   {
        body: form_data,
        headers: headers
   }
);
res.render('index', {access_token: body.access_token});
```

If the request is successful, the authorization server will return a JSON object which includes the access token value, among a few other things. This response looks something like the following:

```
{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer"
}
```

Our application needs to read this result and parse the JSON object to retrieve the access token value, so we parse the response into the `body` variable:

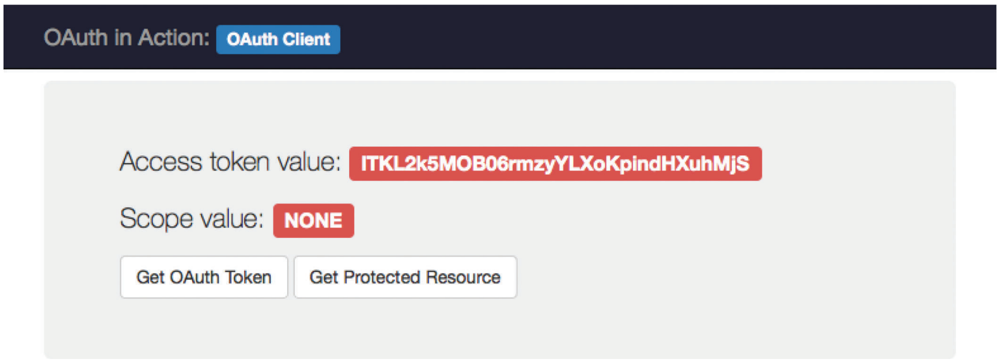
```
var body = JSON.parse(tokRes.getBody());
```

Our client now needs to save this token so that we can use it later:

```
access_token = body.access_token;
```

Our final function for this part of the OAuth client looks like listing 2 in appendix B.

When the access token is successfully fetched and stored, we can send the user back to a page that displays the token value in the browser (figure 3.4). In a real OAuth application, it's a *terrible* idea to display the access token like this since it's a secret value that the client is supposed to be protecting. In our demo application, this helps us visualize what's happening, so we'll let this terrible security practice slide and caution you to be smarter in your production applications.



Access token value: ITKL2k5MOB06rmzyYLXoKpindHXuhMjS

Scope value: NONE

Figure 3.4 The client's homepage after retrieving an access token; the access token value will vary each time the program is run

[redacted]

3.2.3 Adding cross-site protection with the state parameter

In the current setup, any time someone comes to `http://localhost:9000/callback`, the client will naively take in the input `code` value and attempt to post it to the authorization server. This would mean that an attacker could use our client to fish for valid authorization codes at the authorization server, wasting both client and server resources and potentially causing our client to fetch a token it never requested.

We can mitigate this by using an optional OAuth parameter called `state`, which we'll fill with a random value and save to a variable on our application. Right after we throw out our old access token, we'll create this value:

```
state = randomstring.generate();
```

It's important that this value be saved to a place in our application that will still be available when the call to the `redirect_uri` comes back. Remember, since we're using the front channel to communicate in this stage, once we send the redirect to the authorization endpoint, our client application cedes control of the OAuth protocol until this return call happens. We'll also need to add the `state` to the list of parameters sent on the authorization URL.

```
var authorizeUrl = buildUrl(authServer.authorizationEndpoint, {
   response_type: 'code',
   client_id: client.client_id,
   redirect_uri: client.redirect_uris[0],
   state: state
});
```

When the authorization server receives an authorization request with the `state` parameter, it must always return that `state` parameter unchanged to the client alongside the authorization code. This means that we can check the `state` value that's passed in to the `redirect_uri` page and compare it to our saved value from earlier. If it doesn't match, we'll display an error to the end user.

```
if (req.query.state != state) {
   res.render('error', {error: 'State value did not match'});
   return;
}
```

If the `state` value doesn't match what we're expecting, that's a very good indication that something untoward is happening, such as a session fixation attack, fishing for a valid authorization code, or other shenanigans. At this point, the client stops all processing of the request and sends the user to an error page.

3.3 Use the token with a protected resource

Now that we've got an access token, so what? What can we do with it? Fortunately, we've got a handy protected resource running that's waiting for a valid access token and will give us some valuable information when it receives one.

All the client has to do is make a call to the protected resource and include the access token in one of the three valid locations. For our client, we'll send it in the

[redacted]

Authorization HTTP header, which is the method recommended by the specification wherever possible.

Ways to send a bearer token

The kind of OAuth access token that we have is known as a bearer token, which means that whoever holds the token can present it to the protected resource. The OAuth Bearer Token Usage specification actually gives three ways to send the token value:

- As an HTTP Authorization header
- As a form-encoded request body parameter
- As a URL-encoded query parameter

The Authorization header is recommended whenever possible because of limitations in the other two forms. When using the query parameter, the value of the access token can possibly inadvertently leak into server-side logs, because it's part of the URL request. Using the form-encoded parameter limits the input type of the protected resource to using form-encoded parameters and the POST method. If the API is already set up to do that, this can be fine as it doesn't experience the same security limitations that the query parameter does.

The Authorization header provides the maximum flexibility and security of all three methods, but it has the downside of being more difficult for some clients to use. A robust client or server library will provide all three methods where appropriate, and in fact our demonstration protected resource will accept an access token in any of the three locations.

Loading up the index page of our client application from http://localhost:9000/ again, we can see that there's a second button: Get Protected Resource. This button sends us to the data display page.

```
app.get('/fetch_resource', function(req, res){

});
```

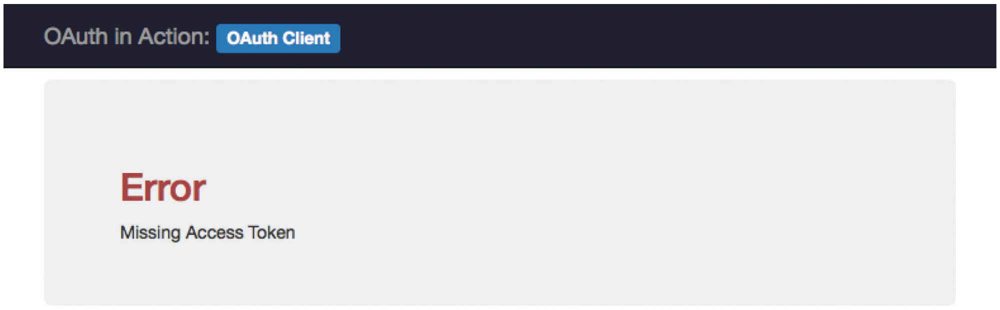
First we need to make sure that we even have an access token. If we don't have one, then we'll display an error to the user and quit.

```
if (!access_token) {
   res.render('error', {error: 'Missing access token.'});
   return;
}
```

If we run this code without getting an access token, we'll get the expected error page, shown in figure 3.5.

In the body of this function we need to call the protected resource and hand its response data off to a page to be rendered. First we need to know where we're going to send the request, and we've configured that in the `protectedResource` variable at the top of the client's code. We'll be making a POST request to that URL and expecting a JSON response back. In other words, this is a very standard API access

[redacted]



Error

Missing Access Token

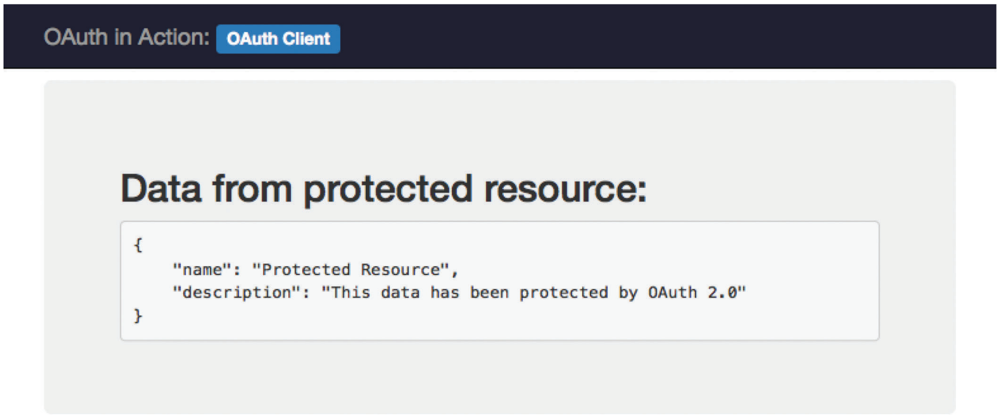
Figure 3.5 An error page on the client, displayed when the access token is missing

request. Nonetheless, it doesn't work yet. Our protected resource is expecting this to be an authorized call, and our client, although capable of getting an OAuth token, isn't yet doing anything with it. We need to send the token using the OAuth-defined `Authorization: Bearer` header, with the token as the value of the header.

```
var headers = {
   'Authorization': 'Bearer ' + access_token
};
var resource = request('POST', protectedResource,
   {headers: headers}
);
```

This sends a request to the protected resource. If it's successful, we'll parse the returned JSON and hand it to our data template. Otherwise, we'll send the user to an error page.

```
if (resource.statusCode >= 200 && resource.statusCode < 300) {
   var body = JSON.parse(resource.getBody());

   res.render('data', {resource: body});
   return;
} else {
```



Data from protected resource:

```
{
    "name": "Protected Resource",
    "description": "This data has been protected by OAuth 2.0"
}
```

Figure 3.6 A page displaying data from the protected resource's API.

[redacted]

```
    res.render('error', {error: 'Server returned response code: ' + resource.
    statusCode});
    return;
}
```

Altogether, our request function looks like listing 3 in appendix B. Now when we obtain an access token and fetch the resource, we get a display of the data from the API (see figure 3.6).

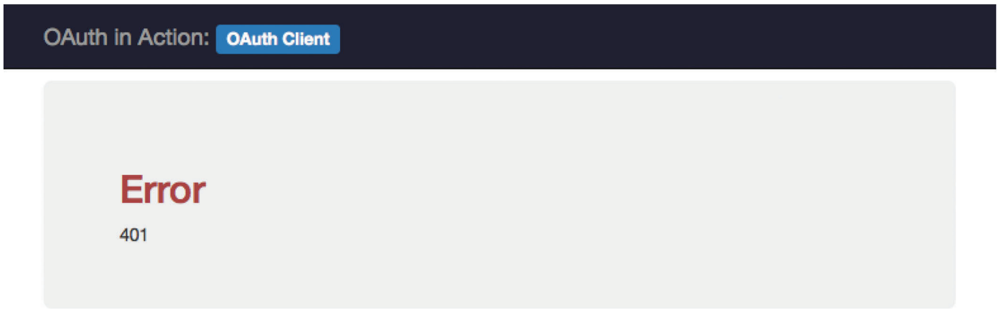
As an additional exercise, try asking the user to authorize the client automatically when the request is made and fails. You could also do this automatic prompting when the client detects that it has no access token to begin with.

3.4 Refresh the access token

Now we've used our access token to fetch a protected resource, but what happens if somewhere down the line our access token expires? Will we need to bother the user again to reauthorize the client application for us?

OAuth 2.0 provides a method of getting a new access token without involving the user interactively: *the refresh token*. This functionality is important because OAuth is often used for cases in which the user is no longer present after the initial authorization delegation has occurred. We covered the nature of refresh tokens in detail in chapter 2, so now we'll be building in support for them to our client.

We'll be starting with a new codebase for this exercise, so open up `ch-3-ex-2` and run `npm install` to get things started. This client has already been set up with an access token and refresh token, but its access token is no longer valid, as if it had expired since it was issued. The client doesn't know that its access token is currently invalid, so it's going to try and use it anyway. This call to the protected resource is going to fail, and we'll be programming the client to use its refresh token to get a new access token and try the call to the protected resource again with the new access token. Run all three applications and open up `client.js` in a text editor. If you want, you can try out the client before we change anything and see that it fails with an invalid token response, indicated by the HTTP error code `401` (see figure 3.7).



Error

401

Figure 3.7 An error page showing the HTTP error code from the protected resource because of an invalid access token

[redacted]

Is my token any good?

How does an OAuth client know whether its access token is any good? The only real way to be sure is to use it and see what happens. If the token is expected to expire, the authorization server can give a hint as to the expected expiration by using the optional `expires_in` field of the token response. This is a value in seconds from the time of token issuance that the token is expected to no longer work. A well-behaved client will pay attention to this value and throw out any tokens that are past the expiration time.

However, knowledge of the expiration alone isn't sufficient for a client to know the status of the token. In many OAuth implementations, the resource owner can revoke the token before its expiration time. A well-designed OAuth client must always expect that its access token could suddenly stop working at any time, and be able to react accordingly.

If you did the extra part of the last exercise, you know that you can prompt the user to authorize the client again and get a new token. This time, though, we've got a refresh token, so if this works we won't have to bug the user at all. The refresh token was originally returned to this client in the same JSON object that the access token came in, like this:

```
{
   "access_token": "987tghjkiu6trfghjuytrghj",
   "token_type": "Bearer",
   "refresh_token": "j2r3oj32r23rmasd98uhjrk2o3i"
}
```

Our client has saved this in the `refresh_token` variable, which we've simulated at the top of the code by setting it to this known value.

```
var access_token = '987tghjkiu6trfghjuytrghj';
var scope = null;
var refresh_token = 'j2r3oj32r23rmasd98uhjrk2o3i';
```

Our authorization server automatically inserts the previous refresh token upon startup, after clearing its database. We don't insert an access token corresponding to the previous one because we want to set up an environment in which the access token has already expired but the refresh token still works.

```
nosql.clear();
nosql.insert({ refresh_token: 'j2r3oj32r23rmasd98uhjrk2o3i', client_id:
'oauth-client-1', scope: 'foo bar' });
```

Now we need to handle refreshing the token. First, we trap the error condition and invalidate our current access token. We'll do this by adding handling to the `else` clause of the code that processes the response from the protected resource.

```
if (resource.statusCode >= 200 && resource.statusCode < 300) {
   var body = JSON.parse(resource.getBody());
```

[redacted]

```
   res.render('data', {resource: body});
   return;
} else {
   access_token = null;
   if (refresh_token) {
       refreshAccessToken(req, res);
       return;
   } else {
       res.render('error', {error: resource.statusCode});
       return;
   }
}
```

Inside the `refreshAccessToken` function, we create a request to the token endpoint, much like we did before. As you can see, refreshing an access token is a special case of an authorization grant, and we use the value `refresh_token` for our `grant_type` parameter. We also include our refresh token as one of the parameters.

```
var form_data = qs.stringify({
   grant_type: 'refresh_token',
   refresh_token: refresh_token
});
var headers = {
   'Content-Type': 'application/x-www-form-urlencoded',
   'Authorization': 'Basic ' + encodeClientCredentials(client.client_id,
   client.client_secret)
};
var tokRes = request('POST', authServer.tokenEndpoint, {
       body: form_data,
       headers: headers
});
```

If the refresh token is valid, the authorization server returns a JSON object, as if this were a normal first-time call to the token endpoint:

```
{
   "access_token": "IqTnLQKcSY62klAuNTVevPdyEnbY82PB",
   "token_type": "Bearer",
   "refresh_token": "j2r3oj32r23rmasd98uhjrk2o3i"
}
```

We can now save the access token value, as we did previously. This response can also include a refresh token, which could be different from the first one. If that happens, the client needs to throw away the old refresh token that it's been saving and immediately start using the new one.

```
access_token = body.access_token;
if (body.refresh_token) {
   refresh_token = body.refresh_token;
}
```

Finally, we can tell our client to try to fetch the resource again. Since our client's actions are triggered by URLs, we can redirect back to the fetch URL and start the

[redacted]

process again. A production implementation will likely have a more sophisticated action trigger.

```
res.redirect('/fetch_resource');
```

To see it working, load up your components and click on Get Protected Resource from the client. Instead of an error, caused by the invalid access token the client boots with, you should see the protected resource data screen. Check out the console for the authorization server: it will indicate that it's issuing a refresh token and will display the values of the tokens used for each request.

```
We found a matching refresh token: j2r3oj32r23rmasd98uhjrk2o3i
Issuing access token IqTnLQKcSY62klAuNTVevPdyEnbY82PB for refresh token
j2r3oj32r23rmasd98uhjrk2o3i
```

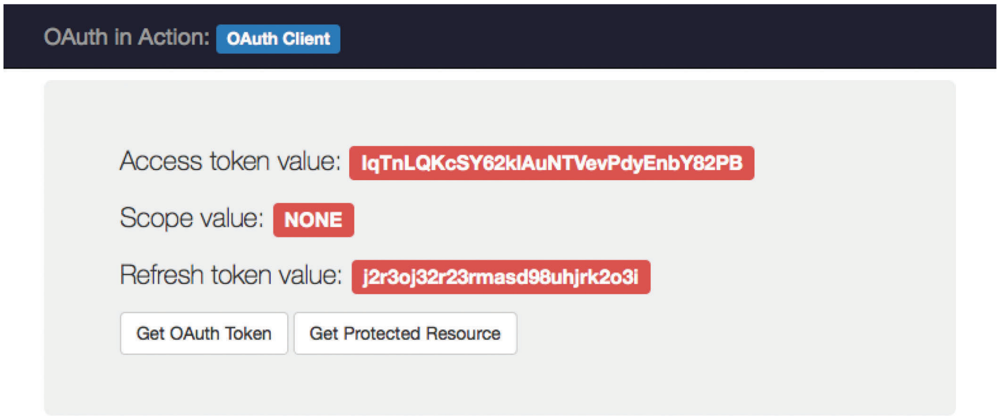
You can also see that the access token's value has changed on the client's homepage by clicking on the title bar of the client application. Compare the access token and refresh token values with those used at the start of the application (figure 3.8).

What happens if the refresh token doesn't work? We throw away both the refresh and access tokens and render an error.

```
} else {
   refresh_token = null;
   res.render('error', {error: 'Unable to refresh token.'});
   return;
}
```

However, we don't need to stop there. Since this is an OAuth client, we're back to the state that we would have been in if we didn't have an access token at all, and we can ask the user to authorize the client again. As an additional exercise, detect this error condition and request a new access token from the authorization server. Be sure to save the new refresh token as well.

The full fetch and refresh functions look like listing 4 in appendix B.


OAuth in Action: OAuth Client

Access token value: IqTnLQKcSY62klAuNTVevPdyEnbY82PB

Scope value: NONE

Refresh token value: j2r3oj32r23rmasd98uhjrk2o3i

Figure 3.8 The client's homepage after refreshing the access token

[redacted]

3.5 Summary

The OAuth client is the most widely used part of the OAuth ecosystem.

- Getting a token using the authorization code grant type requires only a few straightforward steps.
- If it's available to a client, the refresh token can be used to get a new access token without involving the end user.
- Using an OAuth 2.0 bearer token is even simpler than getting one, requiring a simple HTTP header to be added to any HTTP call.

Now that you've seen how the client works, let's build a resource for our client to access.

[redacted]